gen_statem Unveiled
A Theoretical Exploration of State Machines

Nelson Vides
Senior Erlang Consultant and Core MongooseIM developer
nelson.vides@erlang-solutions.com
Who we are

We build massively scalable, distributed systems for high volumes of concurrent users, using resilient and fault-tolerant technologies.

Our team proudly gives back to the community through open source technologies and conferences.

Our work

We work on some of the most famous systems in the world.

Our team

- Globally based
- Diverse & multi-cultural
- United by our passion for technology
Protocols

A system of rules that allows two or more entities of to transmit information.
“The protocol defines the rules, syntax, semantics, and synchronisation of communication and possible error recovery methods.”

Wikipedia: Communication protocol
TCP

- Connection-oriented
- Stream-oriented
- Ordered
- Acknowledged
TLS

— Privacy
— Integrity
— Authenticity
XML or JSON

- File formats
- Structured
HTTP

- Stateless
- Request-response
XMPP

- Extensible
- Instant messaging
- Presence information
- Contact list maintenance
Much like a flowchart on steroids, finite state machines represent an easy way to visualise complex computation flows through a system.
State machines can be seen as models that represent system behaviour
Formalities

- An Alphabet $\Sigma$: a finite set of input symbols $\alpha$
  - $\Sigma = \{0, 1\}; \Sigma = $ [ASCII]; $\Sigma = $ [bounded valid XML]

- A string $\omega$ over $\Sigma$: a finite concatenation of symbols of the alphabet $\Sigma$
  - “0110101” over $\{0, 1\}$

- The Power of an Alphabet: $\Sigma^*$, the (infinite) set of all possible strings over $\Sigma$, including the empty set (an empty string).
  - {“”), {“0”), {“1”), {“”, “0”), {“”, “1”), {“”, “0”, “1”), ...}
Finite State Machines

- an alphabet $\Sigma$, a finite set of states $Q$, an a subset of final states $F$
- and nothing else
- a function $\delta$ such that given states $p \in Q$, an input symbol $\alpha \in \Sigma$:
  - $\delta(p, \alpha) \rightarrow q$, such that $q \in Q$

An FSM is said to consume a string $\omega$ over $Q$ when:

- $\delta(p, \omega) = q$ with $q \in F$

*FSMs are equivalent to regular grammars* (regexes)
Pushdown Automata

- an alphabet $\Sigma$, a finite set of states $Q$, and a subset of final states $F$
- a stack $\Gamma$ of symbols of $\Sigma$
- a function $\delta$ such that given states $p \in Q$, an input symbol $\alpha \in \Sigma$, and the stack $\Gamma$:
  
  $\delta(p, \alpha, \Gamma) \rightarrow (q, \Gamma_2)$ where $q \in Q$ and $\Gamma_2$ is $\Gamma$ after popping the last symbol, pushing a new symbol, or both.

A PDA is said to consume a string $\omega$ over $Q$ when:

- $\delta(p, \omega, \Gamma) = (q, \emptyset)$ with $q \in F$ and $\emptyset$ the empty stack

*PDAs are equivalent to Context-Free Grammars* (parsers)
Turing Machines

- an alphabet \( \Sigma \), a finite set of states \( Q \), and a subset of final states \( F \)
- an infinite tape \( T \) of cells with symbols in \( \Sigma \), and a tape head \( H \)
- a function \( \delta \) such that given states \( p \in Q \), an input symbol \( \alpha \in \Sigma \), and the tape \( T \) with its head \( H \):
  - \( \delta(p, \alpha, H) \rightarrow (q, \beta, H') \) where \( q \in Q \), \( \beta \in \Sigma \) to write in \( H' \), and \( H' \) is the new head of \( T \) after moving one step to left or right and writing a new symbol to it

A Turing Machine is said to consume a string \( \omega \) over \( Q \) when:

- \( \delta(p, \omega, \Gamma) \) is undefined, that is, the machine halts

*TM*s are equivalent to *unrestricted grammars* (compilers)
How do they compare?

Unrestricted Grammars
TM

Context-free Grammars
PDA

Regular Grammars
FSM
How do they compare?

Unrestricted Grammars
- TM
- multi-tape TM, TM with tape bounded on one side

Context-free Grammars
- PDA
- PDA with a finite buffer

Regular Grammars
- FSM
- 2xFSM running together
How do they compare?

Unrestricted Grammars
- TM
- multi-tape TM, TM with tape bounded on one side
- PDA with 2 stacks

Context-free Grammars
- PDA
- PDA with a finite buffer
- PDA + FSM

Regular Grammars
- FSM
- 2xFSM running together
- NxFSM running together
Conceptually, Finite State Machines can keep track of one thing. While Pushdown Automata can keep track of up to two things. While Turing Machines can keep track of a countably infinite number of things.
two, second

dwa, dwie, dwoje, dwóch, dwu, dwaj, dwiema, dwom, dwóm, dwoma, dwojga, dwojgu, dwojgiem, dwojka, dwójki, dwójkę, dwójką, dwójce, dwójko
Do FSMs produce output?
Finite State Transducers

- an alphabet $\Sigma$, a finite set of states $Q$, an a subset of final states $F$
- and an output alphabet $\Lambda$,
- a function $\delta$ such that given states $p \in Q$, an input symbol $\alpha \in \Sigma$:
  - $\delta(p, \alpha) \rightarrow (q, \lambda)$ such that $q \in Q$, $\lambda \in \Lambda$

An FST is said to consume a string $\omega$ over $Q$ when:

- $\delta(p, \omega) = q$ with $q \in F$

FSTs are (also) equivalent to regular grammars (regexes)
\[ \delta(p, \alpha) \rightarrow (q, \lambda) \]

- **Mealy:**
  - Transition: \( \delta_1(p, \alpha) \rightarrow q \in Q \)
  - Output: \( \delta_2(p, \alpha) \rightarrow \lambda \in \Lambda \)

- **Moore:**
  - Transition: \( \delta_1(p, \alpha) \rightarrow q \in Q \)
  - Output: \( \delta_2(p) \rightarrow \lambda \in \Lambda \)
How do they compare?

Turing machines and Pushdown automata

FST
Mealy > Moore

FSM
How do they compare?

- Turing machines and Pushdown automata
- FST
  - Mealy > Moore
  - These can be composed!
- FSM
Composition

- Given sets of states P, Q, R
- And alphabets $\Sigma, \Lambda, \Gamma$

  - Mealy 1: $\delta(P, \Sigma) \rightarrow (Q, \Lambda)$
  - Mealy 2: $\delta(Q, \Lambda) \rightarrow (R, \Gamma)$
  - Mealy Composition: $\delta(P, \Sigma) \rightarrow (R, \Gamma)$
An algebra

Given FSMs $M_1$ and $M_2$, the following are true:

- $(M_1 \cup M_2)$ is an FSM
- $(M_1 \cap M_2)$ is an FSM
- $(M_1 \cdot M_2)$ is an FSM
- The reverse, and the inverse, are also FSMs
- $\emptyset$ is the neutral element under union and concatenation
- Homomorphisms also preserve FSMs

FSMs form a semiring under union and concatenation
Why is such algebra useful?

Proving theorems!
gen_statem: $\delta(P, \Sigma) \rightarrow (Q, \Lambda)$

- $P$ and $Q$ are the states as implemented by the programmer
- $\Sigma$ are messages in the mailbox
- $\Lambda$ are side-effects
An extended mailbox
A mental picture

- Only event one queue that is an extension of the process mailbox.
- This queue has got:
  - A head pointing at the oldest event;
  - A current pointing at the next event to be processed.
  - A tail pointing at the youngest event;
A mental picture

- **Postpone** causes the current position to move to its next younger event so the previous current position is still in the queue reachable from head.

- **Not postponing** an event i.e consuming it causes the event to be removed from the queue and current position to move to its next younger event.

- **NewState != State** causes the current position to be set to head i.e the oldest event.

- **next_event** inserts event(s) at the current position i.e as just older than the previous current position.

- **{timeout, 0, Msg}** inserts a {timeout, Msg} event after tail i.e as the new youngest received event.
The event queue

```prolog
01 handle_event(Type1, Content1, State1, Data) ->
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
The event queue

01 handle_event(Type1, Content1, State1, Data) ->
02 {keep_state_and_data, [postpone]};
The event queue

handle_event(Type1, Content1, State1, Data) ->
{keep_state_and_data, [postpone]};
handle_event(Type1, Content1, State1, Data) ->
  {keep_state_and_data, [postpone]};
... handle_event(Type2, Content2, State1, Data) ->
01 handle_event(Type1, Content1, State1, Data) ->
02   {keep_state_and_data, [postpone]};
03 ...
04 handle_event(Type2, Content2, State1, Data) ->
05   {next_state, State2};
The event queue

01 handle_event(Type1, Content1, State1, Data) ->
02     {keep_state_and_data, [postpone]};
03 ... 
04 handle_event(Type2, Content2, State1, Data) ->
05     {next_state, State2};
The event queue

01 handle_event(Type1, Content1, State1, Data) ->
02 {keep_state_and_data, [postpone]};
03 ...
04 handle_event(Type2, Content2, State1, Data) ->
05 {next_state, State2};
06 ...
07 handle_event(Type1, Content1, State2, Data) ->
01  handle_event(Type1, Content1, State1, Data) ->
02      {keep_state_and_data, [postpone]};
03  ...
04  handle_event(Type2, Content2, State1, Data) ->
05      {next_state, State2};
06  ...
07  handle_event(Type1, Content1, State2, Data) ->
08      {keep_state_and_data, [{next_event, TypeA, ContentA}]};

The event queue

```
01  handle_event(Type1, Content1, State1, Data) ->
02      {keep_state_and_data, [postpone]};
03  ...
04  handle_event(Type2, Content2, State1, Data) ->
05      {next_state, State2};
06  ...
07  handle_event(Type1, Content1, State2, Data) ->
08      {keep_state_and_data, [{next_event, TypeA, ContentA}]};
```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};

The event queue

Type_A, Content_A
Type_3, Content_3
Type_4, Content_4
Type_5, Content_5
The event queue

```
01 handle_event(Type1, Content1, State1, Data) ->
02     {keep_state_and_data, [postpone]};
03 ...
04 handle_event(Type2, Content2, State1, Data) ->
05     {next_state, State2};
06 ...
07 handle_event(Type1, Content1, State2, Data) ->
08     {keep_state_and_data, [{next_event, TypeA, ContentA}]};
09 ...
10 handle_event(TypeA, ContentA, State2, Data) ->
11     {keep_state_and_data, [postpone]};
```
handle_event(Type1, Content1, State1, Data) ->
{keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
{next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
{keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
{keep_state_and_data, [postpone]};
handle_event(Type1, Content1, State1, Data) ->
  {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
  {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
  {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
  {keep_state_and_data, [postpone]};
...
handle_event(Type3, Content3, State2, Data) ->
handle_event(Type1, Content1, State1, Data) ->
  {keep_state_and_data, [postpone]};
handle_event(Type2, Content2, State1, Data) ->
  {next_state, State2};
handle_event(Type1, Content1, State2, Data) ->
  {keep_state_and_data, [{next_event, TypeA, ContentA}]};
handle_event(TypeA, ContentA, State2, Data) ->
  {keep_state_and_data, [postpone]};
handle_event(Type3, Content3, State2, Data) ->
  keep_state_and_data;
The event queue

01  handle_event(Type1, Content1, State1, Data) ->
02      {keep_state_and_data, [postpone]};
03  ...
04  handle_event(Type2, Content2, State1, Data) ->
05      {next_state, State2};
06  ...
07  handle_event(Type1, Content1, State2, Data) ->
08      {keep_state_and_data, [{next_event, TypeA, ContentA}]};
09  ...
10  handle_event(TypeA, ContentA, State2, Data) ->
11      {keep_state_and_data, [postpone]};
12  ...
13  handle_event(Type3, Content3, State2, Data) ->
14      keep_state_and_data;
01 handle_event(Type1, Content1, State1, Data) ->
02   {keep_state_and_data, [postpone]};
03   ...
04 handle_event(Type2, Content2, State1, Data) ->
05   {next_state, State2};
06   ...
07 handle_event(Type1, Content1, State2, Data) ->
08   {keep_state_and_data, [{next_event, TypeA, ContentA}]};
09   ...
10 handle_event(TypeA, ContentA, State2, Data) ->
11   {keep_state_and_data, [postpone]};
12   ...
13 handle_event(Type3, Content3, State2, Data) ->
14   keep_state_and_data;
15   ...
16 handle_event(Type4, Content4, State2, Data) ->
The event queue

```
01 handle_event(Type1, Content1, State1, Data) ->
02   {keep_state_and_data, [postpone]};
03 ...
04 handle_event(Type2, Content2, State1, Data) ->
05   {next_state, State2};
06 ...
07 handle_event(Type1, Content1, State2, Data) ->
08   {keep_state_and_data, [{next_event, TypeA, ContentA}]};
09 ...
10 handle_event(TypeA, ContentA, State2, Data) ->
11   {keep_state_and_data, [postpone]};
12 ...
13 handle_event(Type3, Content3, State2, Data) ->
14   keep_state_and_data;
15 ...
16 handle_event(Type4, Content4, State2, Data) ->
17   {keep_state_and_data,
18    [postpone, {next_event, TypeB, ContentB}]};
```
The event queue

```erlang
handle_event(Type1, Content1, State1, Data) ->
  {keep_state_and_data, [postpone]};

handle_event(Type2, Content2, State1, Data) ->
  {next_state, State2};

handle_event(Type1, Content1, State2, Data) ->
  {keep_state_and_data, [{next_event, TypeA, ContentA}]};

handle_event(TypeA, ContentA, State2, Data) ->
  {keep_state_and_data, [postpone]};

handle_event(Type3, Content3, State2, Data) ->
  keep_state_and_data;

handle_event(Type4, Content4, State2, Data) ->
  {keep_state_and_data, [postpone, {next_event, TypeB, ContentB}]};
```

The event queue:
- HEAD: Type_A, Content_A
- CURR: Type_4, Content_4
- TAIL: Type_5, Content_5
A mental picture

Only event one queue that is an extension of the process mailbox, with:

- A *head* pointing at the oldest event;
- A *current* pointing at the next event to be processed.
- A *tail* pointing at the youngest event;
Managing accidental complexity
An archetypical example
handle_call(on, _From, {off, Light}) ->
  on = request(on, Light),
  {reply, on, {on, Light}};
handle_call(off, _From, {on, Light}) ->
  off = request(off, Light),
  {reply, off, {off, Light}};
handle_call(on, _From, {on, Light}) ->
  {reply, on, {on, Light}};
handle_call(off, _From, {off, Light}) ->
  {reply, off, {off, Light}}.

All requests to the light are synchronous
handle_call(on, From, {off, undefined, Light}) ->
    Ref = request(on, Light),
    {noreply, {off, {on, Ref, From}, Light}}.
handle_call(off, From, {on, undefined, Light}) ->
    Ref = request(off, Light),
    {noreply, {on, {off, Ref, From}, Light}}.

handle_call(off, _From, {on, {off, _, _}, Light} = State) ->
    {reply, turning_off, State}.  %% ???
handle_call(on, _From, {off, {on, _, _}, Light} = State) ->
    {reply, turning_on, State}.  %% ???
handle_call(off, _From, {off, {on, _, _}, Light} = State) ->
    {reply, turning_on_wait, State}.  %% ???
handle_call(on, _From, {on, {off, _, _}, Light} = State) ->
    {reply, turning_off_wait, State}.  %% ???

handle_info(Ref, {State, {Request, Ref, From}, Light}) ->
    gen_server:reply(From, Request),
    {noreply, {Request, undefined, Light}}.

What do we do with queueing requests now? **Careful with user-level callstacks!**
off({call, From}, off, {undefined, Light}) ->
    {keep_state_and_data, [{reply, From, off}]};
off({call, From}, on, {undefined, Light}) ->
    Ref = request(on, Light),
    {keep_state, [{Ref, From}, Light], []};
off({call, From}, _, _) ->
    {keep_state_and_data, [postpone]};
off(info, {Ref, Response}, {{Ref, From}, Light}) ->
    {next_state, Response, {undefined, Light}, [{reply, From, Response}]}.

11 on({call, From}, on, {undefined, Light}) ->
    {keep_state_and_data, [{reply, From, on}]};
on({call, From}, off, {undefined, Light}) ->
    Ref = request(off, Light),
    {keep_state, [{Ref, From}, Light], []};
on({call, From}, _, _) ->
    {keep_state_and_data, [postpone]};
on(info, {Ref, Response}, {{Ref, From}, Light}) ->
    {next_state, Response, {undefined, Light}, [{reply, From, Response}]}.

Try with a gen_statem... cool, we can “postpone” events! But it feels repetitive, there is **no code reuse**.
Apparent problems

- There is no global ordering
- Tying yourself to the actual ordering of events, leads to accidental complexity
- Complexity grows relative to the number of possible permutations of event sequences... unless you have a strategy for “reordering events”
- Code reuse becomes practically impossible

Ulf Wiger: Death by accidental complexity
In a nutshell

define

handle_event({call, From}, State, State, {undefined, Light}) ->
    {keep_state_and_data, [{reply, From, State}]};

handle_event({call, From}, Request, State, {undefined, Light}) ->
    Ref = request(Request, Light),
    {keep_state, {{Ref, From}, Light}, []};

handle_event({call, _}, _, _, _) ->
    {keep_state_and_data, [postpone]};

handle_event(info, {Ref, Response}, State, {{Ref, From}, Light}) ->
    {next_state, Response, {undefined, Light}, [{reply, From, Response}]}.
“With great power comes great responsibility.”
This is my gift, my curse.
A case study

XMPP server and client state machines
01  wait_for_stream(stream_start, Data#{auth = false}) ->
02    {next, wait_for_feature};
03  wait_for_stream(stream_start, Data#{auth = true}) ->
04    {next, wait_for_feature};
05
06  wait_for_feature(authenticate, Data#{auth = false}) ->
07    {next, wait_for_stream, Data#{auth = true}};
08  wait_for_feature(session, Data#{auth = true}) ->
09    {next, session_established};
10
11  wait_for_sasl_response(auth_response, Data) ->
12    {next, wait_for_stream, Data#data{auth = true}};
13
14  session_established(Event, Data) ->
15    {next, session_established};
16
17  ...  
18
19  handle_info({tls, S, D}, Data#{socket = S}) ->
20    Decrypt = tls:decrypt(D),
21    XmlEl = exml:parse(Decrypt),
22    Fsm ! {xml, XmlEl}.
23
24  handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
25    S ! XmlEl;
```erlang
wait_for_stream(stream_start, Data#{auth = false}) ->
  {next, wait_for_feature};
wait_for_stream(stream_start, Data#{auth = true}) ->
  {next, wait_for_feature};
wait_for_feature(authenticate, Data#{auth = false}) ->
  {next, wait_for_stream, Data#{auth = true}};
wait_for_feature(session, Data#{auth = true}) ->
  {next, session_established};
wait_for_sasl_response(auth_response, Data) ->
  {next, wait_for_stream, Data#{data = true}};
session_established(Event, Data) ->
  {next, session_established};
handle_info({tls, S, D}, Data#{socket = S}) ->
  Decrypt = tls:decrypt(D),
  XmlEl = exml:parse(Decrypt),
  Fsm ! {xml, XmlEl}.
handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
  S ! XmlEl;
```
wait_for_stream(stream_start, Data#{auth = false}) ->
    {next, wait_for_feature};
wait_for_stream(stream_start, Data#{auth = true}) ->
    {next, wait_for_feature};

wait_for_feature(authenticate, Data#{auth = false}) ->
    {next, wait_for_stream, Data#{auth = true}};

wait_for_feature(session, Data#{auth = true}) ->
    {next, session_established};

wait_for_sasl_response(auth_response, Data) ->
    {next, wait_for_stream, Data#data{auth = true}};

session_established(Event, Data) ->
    {next, session_established};

...
wait_for_stream(stream_start, Data#{auth = false}) ->
  {next, wait_for_feature};

wait_for_stream(stream_start, Data#{auth = true}) ->
  {next, wait_for_feature};

wait_for_feature(authenticate, Data#{auth = false}) ->
  {next, wait_for_stream, Data#{auth = true}};

wait_for_feature(session, Data#{auth = true}) ->
  {next, session_established};

wait_for_sasl_response(auth_response, Data) ->
  {next, wait_for_stream, Data#data{auth = true}};

session_established(Event, Data) ->
  {next, session_established};

handle_info({tls, S, D}, Data#{socket = S}) ->
  Decrypt = tls:decrypt(D),
  XmlEl = exml:parse(Decrypt),
  Fsm ! {xml, XmlEl}.

handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
  S ! XmlEl;
```erlang
wait_for_stream(stream_start, Data#{auth = false}) ->
    {next, wait_for_feature};
wait_for_stream(stream_start, Data#{auth = true}) ->
    {next, wait_for_feature};

wait_for_feature(authenticate, Data#{auth = false}) ->
    {next, wait_for_stream, Data#{auth = true}};
wait_for_feature(session, Data#{auth = true}) ->
    {next, session_established};
wait_for_sasl_response(auth_response, Data) ->
    {next, wait_for_stream, Data#{data = true}};

session_established(Event, Data) ->
    {next, session_established};

handle_info({tls, S, D}, Data#{socket = S}) ->
    Decrypt = tls:decrypt(D),
    XmlEl = exml:parse(Decrypt),
    Fsm ! {xml, XmlEl}.
handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
    S ! XmlEl;
```

Finite State Machine

We are using an automaton more powerful than is really needed!
Authenticated

Server

TCP open

wait for stream

stream start

wait for features

auth

SASL

response

wait for sasl response

Unauthenticated

Client

TCP open

wait for stream

stream

features

SASL?

wait for sasl challenge

response

Unauthenticated

wait for stream and features

session

session established

Authenticated

TCP close

wait for features

session

session established

TCP close

wait for sasresponse

stanzas
Complex States

XMPP client

```erlang
handle_event(Type, stream, [stream, features, sasl_ch, wait_for, AuthData, not_auth], _)  
handle_event(Type, features, [features, sasl_ch, wait_for, AuthData, not_auth], _)  
handle_event(Type, sasl_ch, [sasl_ch, wait_for, AuthData, not_auth], _)  
handle_event(Type, stream, [stream, features, wait_for, Sess, auth], _)  
handle_event(Type, features, [features, wait_for, Sess, auth], _)  
handle_event(Type, Content, [session_established, Sess, auth], _)  
```
Complex States

XMPP server

```erlang
handle_event(Type, stream, [before_auth, wait_for, stream], _)
handle_event(Type, auth, [before_auth, wait_for, features, Retries, AuthData], _)
handle_event(Type, sasl_ch, [during_auth, wait_for, sasl_response, Retries, AuthData], _)
handle_event(Type, stream, [auth, wait_for, stream], _)
handle_event(Type, session, [auth, wait_for, features, Features], _)
handle_event(Type, Content, [auth, session_established], _)
```
Complex Events

XMPP server

```erlang
handle_event(info, {tcp, Socket, Payload}, StateName, #{parser = P, crypto = C} = StateData) ->
    {P1, C1, XmlElements} = decode(P, decrypt(C, Payload)),
    StreamEvents = [{next_event, internal, El} || #xmlel{} = El <- XmlElements],
    {next_state, StateName, StateData#{parser = P1, crypto = C1}, StreamEvents};
```
Nelson Vides
Senior Erlang Consultant and Core MongooseIM developer
nelson.vides@erlang-solutions.com
gen_statem Unveiled

Questions?